Appendix 1: Source Code for Sample Implementation of Access Delay Reduction Algorithm

This section contains sample C++ source code for a floating-point version of the Access Delay Reduction algorithm. 5 files are listed.

File	Description
pseudocode.c	Pseudo C++ code that shows how to call the ADR algorithm in an application. No implementations are given for many of the functions called in this code as they are system dependent.
adr.h	Header file for the Access Delay Reduction algorithm. Includes declarations for both the public and private parts of the class. Internally, this class uses the CircularBuffer class.
adr.c	Implementation of the Access Delay Reduction algorithm. This file contains the heart of the algorithm and is the most important file included here.
circularbuffer.h	Include file for a circular First In First Out (FIFO) buffer. The CircularBuffer is used internally by adr.c. It is not called directly by the user and is included to clarify its use by the AccessDelayReduducer class.
circularbuffer.c	Implementation of the circular buffer. This file includes "*libcoder.h*" which is not shown here. The only function declared in *libcoder.h* is the *error* function, which halts the system on catastrophic errors.

```
File: pseudocode.c
/*
 * Copyright (c) 1999-2000 AT&T Corp.
 * All Rights Reserved.
 */
#include <circularbuffer.h>
#include <vad.h>
#include <adr.h>

/*
 * pseudo code for main processing loop with Access Delay Reduction algorithm.
 * Read a frame's worth of audio, give it to both the VAD and ADR. When
 * the VAD detects onset of activity, request a transmission channel. In
 * the mean time the ADR buffers the speech. After the access delay, the
 * ADR time-scales the beginning of the talkspurt until the access delay
 * is gone. At the end of the talkspurt, the transmit channel is freed.
 */
void processloop()
{
   int              framesz = 160;      /* 20 msec at 8 KHz */
   Float            y[160];
   bool             activity, oldactivity = false;
   bool             adrdata, oldadrdata = false;
22    Vad                     vad(8000, 160);
23    AccessDelayReducer  adr(8000, 20., 60., 500.);

24    while (readinputframe(y, framesz)) {
25            activity = vad.activity(y);
26            /* request transmission channel at activity onset */
27            if (activity && !oldactivity)
28                 request_tx_channel();
29            adrdata = adr.newframe(y, y, activity);
30            if (adrdata)
31                 encode_and_xmit(y, framesz);
32            /* free channel when ADR buffer has drained */
33            if (!adrdata && oldadrdata)
34                 free_tx_channel();
35            oldactivity = activity;
36            oldadrdata = adrdata;
37    }
38 }
```

Pseudocode.c

The function *processloop* in pseudocode.c shows how the AccessDelayReducer class is used in an application. Here, we have decided to process the speech in increments of 160 samples, or 20 msec at 8 KHz sampling. On line 19 an array large enough to hold one frame's worth of floats is declared. The "Float" type is defined as a float with a *typedef* in the file circularbuffer.h. The *bool*s on lines 20 and 21 keep track of the current and previous state of both the VAD and the ADR. An inactive to active transition detected by the VAD is used to request a transmission channel on lines 27 and 28. On lines 33-34, the end of available data for a talkspurt is used to relinquish the transmission channel. The constructor for the VAD on line 22 sets the VAD frame size to 160 samples and the samplerate to 8 KHz. The constructor call to the AccessDelayReducer on line 23 sets the samplerate to 8 KHz, the frame size to 20 msec, the access delay to 60 msec, and the interval for the time-scaling to 500 msec.

The loop on lines 24-37 reads in a frame of data and processes it. First, the VAD determines if there is activity on line 25. Next the frame is given to the ADR on line 29. The first argument is the input frame and the second argument is the output frame. In this example, the output from the ADR is placed in the same buffer used for input. The speech is buffered and delayed internally by the ADR. The call to *newframe* returns true if the output frame contains speech that should be transmitted (there is activity in it) and false otherwise. At the first few frames after an inactive to active transition in the VAD, e.g. for the duration of the access delay, *newframe* returns false even though the input frames contain active speech. After the access delay is over, the speech at the start of the talkspurt is returned. *Newframe* then starts time-scale compressing the speech until the access delay is removed.

Since the ADR may leave some residual delay or the talkspurt may be too short for the ADR to finish time-scaling, the output of the ADR determines when the transmission channel is returned rather than the VAD. All the active speech buffered in the ADR must be output before channel is returned.

```
File: adr.h
  1 /*
```

```
 * Copyright (c) 1999-2000 AT&T Corp.
 * All Rights Reserved.
 *
 * Performing time-scaling compression at the start of a talkspurt
 * in systems where there is access delay for channel allocation such
 * as Voice over EDGE.
 */
class AccessDelayReducer {
public:
          AccessDelayReducer(int srate, Float framesizems,
                 Float accessdelayms, Float timescaleintervalms);
          ~AccessDelayReducer();
   bool   newframe(Float *in, Float *out, bool active);
protected:
   Float  frameszmsec;  /* frame size in msec */
   Float  sysdelaymsec; /* system contention delay, msec */
   Float  timescalemsec;      /* interval for timescaling, msec */
   Float  targetaccum;  /* target accumulator, samples */
   Float  targetincr;   /* target increment, samples */
   int    samplerate;   /* samplerate, Hz */
   int    framesz;      /* frame size in samples */
   int    activelen;    /* frames in current talk spurt */
   int    sysdelayf;    /* system contention delay, frames */
   int    sysdelay;     /* system delay, samples */
   int    curdelay;     /* current delay, samples */
   int    targetdelay;  /* target delay, samples */
   int    timescalef;   /* timescaling interval, frames */
   int    timescalefirstf;/* first frame to start timescaling */
   int    timescalelastf;     /* last frame to start timescaling */
   int    ndec;         /* decimation factor */
   int    pitchmin;     /* minimum pitch */
   int    pitchmax;     /* maximum pitch */
   int    pitchdiff;    /* pitch difference */
   int    corrlen;      /* correlation length */
   int    corrbuflen;   /* length of correlation buffer */
   CircularBuffer       *outbuf;/* output buffer */
   Float  *tmpbuf;      /* temporary scratch buffer */
   Float  *corrbuf;     /* input buffer */

   int    findbestmatch();
   void   updatecorrbuf(Float *s);
   void   removedelay(Float *in, int pitch);
   void   overlapadd(Float *l, Float *r, Float *o, int cnt);
   void   idle();
   void   copy(Float *f, Float *t, int cnt);
   void   zero(Float *s, int cnt);
};
```

File: adr.c

```
/*
 * Copyright (c) 1999-2000 AT&T Corp.
 * All Rights Reserved.
 */
#include <math.h>
#include "circularbuffer.h"
#include "adr.h"

#define  PITCH_MIN     .0025        /* minimum allowed pitch, 400 Hz */
#define  PITCH_MAX     .015         /* maximum allowed pitch, 66 Hz */
#define  NDEC_8K       2            /* 2:1 decimation at 8kHz */
#define  CORRMINPOWER  ((Float)250.) /* minimum power */
#define  CORRLEN       .020         /* 20 msec correlation length */
```

```
/*
 * Constructor sets the samplerate, the frame size, the estimated access delay
 * and the time-scaling interval. Appropriate length buffers are allocated
 * based on these parameters.
 */
AccessDelayReducer::AccessDelayReducer(int srate, Float framesizems,
   Float accessdelayms, Float timescaleintervalms)
{
   samplerate = srate;
   frameszmsec = framesizems;
   sysdelaymsec = accessdelayms;
   timescalemsec = timescaleintervalms;
   ndec = (int)(NDEC_8K * samplerate / 8000.);
   pitchmin = (int)(PITCH_MIN * samplerate);
   pitchmax = (int)(PITCH_MAX * samplerate);
   pitchdiff = pitchmax - pitchmin;
   corrlen = (int)(CORRLEN * samplerate);
   corrbuflen = corrlen + pitchmax;
   framesz = (int)(samplerate * frameszmsec * (Float).001);
   sysdelayf = (int)ceil(sysdelaymsec / frameszmsec);
   sysdelay = sysdelayf * framesz;
   timescalef = (int)ceil(timescalemsec / frameszmsec) + 1;
   timescalefirstf = sysdelayf + 1;
   timescalelastf = sysdelayf + timescalef;
   targetincr = (Float)sysdelay / (timescalef + 1);
   corrbuf = new Float[corrbuflen];
   outbuf = new CircularBuffer(framesz * (sysdelayf + 1));
   tmpbuf = new Float[pitchmax >> 2];
   activelen = 0;
   idle();
}

/*
 * Free allocated resources in destructor.
 */
AccessDelayReducer::~AccessDelayReducer()
{
   delete [] tmpbuf;
   delete outbuf;
   delete [] corrbuf;
}

/*
 * main public function for time-scaling sppech at start of talkspurt.
 * Input is the speech from the audio port and active indicator from the
 * VAD. Output is the speech delayed by the access delay, and then time-scaled
 * to get remove the delay at the start of the talksprt.
 * Newframe returns true if the returned frame should be transmitted and
 * false if it should not be transmitted. For simulation purposes the
 * returned frame of speech is set to zero if it should not be transmitted.
 */
bool AccessDelayReducer::newframe(Float *in, Float *out, bool active)
{
   bool   r;
   int    pitch, cnt;

   updatecorrbuf(in);
   if (active) {
          /* simulate contention delay at start of utterance */
          if (++activelen <= sysdelayf) {
                 /*
                  * if delayed samples still left from last utterance
                  * flush it. This shouldn't happen since if there
```

```
                         * is some leftover delay, it should be output at
                         * the first frame where the VAD determines there is
                         * no activity.
                         */
                        if (activelen == 1 && outbuf->filled())
                                outbuf->flush();
                        outbuf->write(in, framesz);
                        curdelay += framesz;
                        zero(out, framesz);
                        r = false;
                }
                /* time-scale at start of utterance */
                else {
                        /* update the current amount we allow to timescale */
                        if (activelen <= timescalelastf) {
                                /*
                                 * boost at first frame so targetaccum is
                                 * greater than pitchmin so its possible
                                 * to time-scale at frame timescalefirstf.
                                 */
                                if (activelen == timescalefirstf)
                                        targetaccum = (Float)2. * targetincr;
                                else
                                        targetaccum += targetincr;
                                targetdelay = (int)targetaccum;
                                if (targetdelay > curdelay)
                                        targetdelay = curdelay;
                        }
                        /*
                         * if the target for delay removal is larger than
                         * the minimum pitch, we can try to remove the delay.
                         * We still may not be able to do it yet if the
                         * estimated pitch is larger than the target delay.
                         */
                        if (targetdelay >= pitchmin &&
                             (pitch = findbestmatch()) <= targetdelay) {
                                removedelay(in, pitch);
                                outbuf->read(out, framesz);
                        )
                        /*
                         * either time-scaling isn't necessary, or not
                         * possible because not enough time has passed,
                         * or the current pitch is too long.
                         * If outcnt is 0, all the delay has been removed
                         * so we just copy the data from input to output.
                         * Otherwise, there is still delay in the system
                         * so the output must be buffered.
                         */
                        else if (outbuf->filled() == 0)
                                copy(in, out, framesz);
                        else {
                                outbuf->write(in, framesz);
                                outbuf->read(out, framesz);
                        }
                        r = true;
                }
        }
        /* no speech activity detected */
        else {
                if (activelen != 0) {
                        activelen = 0;
                        idle();
                }
        /* if something left in delay buffer, output it */
        cnt = outbuf->filled();
        if (cnt) {
             if (cnt >= framesz)
                  cnt = framesz;
             int left = framesz - cnt;
             outbuf->read(out, cnt);
             zero(&out[cnt], left);
             if (outbuf->filled() == 0)
                  idle();
             r = true;
        } else {
             zero(out, framesz);
             r = false;
        }
   }
   return r;
}

/* remove the delay by time-scale compressing the input */
void AccessDelayReducer::removedelay(Float *in, int pitch)
{
   int    p2, pq, cnt, olacnt, ocnt;

   /* see if we can remove more than one pitch period at a time */
   p2 = pitch << 1;
   if (p2 <= targetdelay && p2 <= pitchmax)
        pitch = p2;
   pq = pitch >> 2;
   olacnt = pitch + pq;
   /* if the OLA fits in one frame, work directly on the input frame */
   if (olacnt <= framesz) {
        cnt = framesz - olacnt;
        outbuf->write(in, cnt);
        overlapadd(&in[cnt], &in[cnt+pitch], tmpbuf, pq);
        outbuf->write(tmpbuf, pq);
   }
   /* Otherwise we have to copy some samples from the previous frame */
   else {
        cnt = olacnt - framesz;
        ocnt = pq - cnt;
        outbuf->peektail(tmpbuf, cnt);    /* from previous frame tail */
        copy(in, &tmpbuf[cnt], ocnt);     /* from current frame */
        overlapadd(tmpbuf, &in[framesz - pq], tmpbuf, pq);
        outbuf->replacetail(tmpbuf, cnt); /* replace old tail */
        outbuf->write(tmpbuf, ocnt);      /* write tail of OLA */
   }
   /* update the current delay variables */
   targetaccum -= (Float)pitch;
   targetdelay -= pitch;
   curdelay -= pitch;
}

/* Initialized the time-scaling variables */
void AccessDelayReducer::idle()
{
   curdelay = 0;
   targetdelay = 0;
   targetaccum = 0.;
}

/* Save a frames worth of new speech into the correlation buffer */
void AccessDelayReducer::updatecorrbuf(Float *s)
```

```
{
  int offset = corrbuflen - framesz;
  /* make room for new speech frame */
  copy(&corrbuf[corrbuflen - offset], corrbuf, offset);
  /* copy in the new frame */
  copy(s, &corrbuf[offset], framesz);
}

/*
 * Find the best match between the last segment of speech and
 * the previous speech in the correlation buffer.
 */
int AccessDelayReducer::findbestmatch()
{
  int    i, j, k;
  int    bestmatch;
  Float  bestcorr;
  Float  corr;         /* correlation */
  Float  energy;              /* running energy */
  Float  scale;        /* scale correlation by average power */
  Float  *rp;          /* segment to match */
  Float  *l;

  l = &corrbuf[pitchmax];
  /* coarse search */
  rp = corrbuf;
  energy = 0.f;
  corr = 0.f;
  for (i = 0; i < corrlen; i += ndec) {
         energy += rp[i] * rp[i];
         corr += rp[i] * l[i];
  }
  scale = energy;
  if (scale < CORRMINPOWER)
         scale = CORRMINPOWER;
  corr /= (Float)sqrt(scale);
  bestcorr = corr;
  bestmatch = 0;
  for (j = ndec; j <= pitchdiff; j += ndec) {
         energy -= rp[0] * rp[0];
         energy += rp[corrlen] * rp[corrlen];
         rp += ndec;
         corr = 0.f;
         for (i = 0; i < corrlen; i += ndec)
                corr += rp[i] * l[i];
         scale = energy;
         if (scale < CORRMINPOWER)
                scale = CORRMINPOWER;
         corr /= (Float)sqrt(scale);
         if (corr >= bestcorr) {
                bestcorr = corr;
                bestmatch = j;
         }
  }
  /* fine search */
  j = bestmatch - (ndec - 1);
  if (j < 0)
         j = 0;
  k = bestmatch + (ndec - 1);
  if (k > pitchdiff)
         k = pitchdiff;
  rp = &corrbuf[j];
  energy = 0.f;
	corr = 0.f;
	for (i = 0; i < corrlen; i++) {
		energy += rp[i] * rp[i];
		corr += rp[i] * l[i];
	}
	scale = energy;
	if (scale < CORRMINPOWER)
		scale = CORRMINPOWER;
	corr = corr / (Float)sqrt(scale);
	bestcorr = corr;
	bestmatch = j;
	for (j++; j <= k; j++) {
		energy -= rp[0] * rp[0];
		energy += rp[corrlen] * rp[corrlen];
		rp++;
		corr = 0.f;
		for (i = 0; i < corrlen; i++)
			corr += rp[i] * l[i];
		scale = energy;
		if (scale < CORRMINPOWER)
			scale = CORRMINPOWER;
		corr /= (Float)sqrt(scale);
		if (corr > bestcorr) {
			bestcorr = corr;
			bestmatch = j;
		}
	}
	return pitchmax - bestmatch;
}

/* Overlap add with triangular windows */
void AccessDelayReducer::overlapadd(Float *l, Float *r, Float *o, int cnt)
{
	Float incr = (Float)1. / cnt;
	Float lw = (Float)1. - incr;
	Float rw = incr;
	for (int i = 0; i < cnt; i++) {
		o[i] = lw * l[i] + rw * r[i];
		lw -= incr;
		rw += incr;
	}
}

void AccessDelayReducer::copy(Float *f, Float *t, int cnt)
{
	for (int i = 0; i < cnt; i++)
		t[i] = f[i];
}

void AccessDelayReducer::zero(Float *s, int cnt)
{
	for (int i = 0; i < cnt; i++)
		s[i] = (Float)0.;
}
```

File: circularbuffer.h

```
/*
 * Copyright (c) 1999-2000 AT&T Corp.
 * All Rights Reserved.
 *
 * Circular buffer
 */
```

```
typedef float Float;

class CircularBuffer {
public:
            CircularBuffer(int sz);
            ~CircularBuffer();
    void    read(Float *f, int sz);
    void    write(Float *f, int sz);
    void    peekhead(Float *f, int sz);
    void    peektail(Float *f, int sz);
    void    replacehead(Float *f, int sz);
    void    replacetail(Float *f, int sz);
    void    flush();
    void    clear();
    int     capacity()    { return buflen; }
    int     filled()      { return cnt; }
protected:
    int     buflen;               /* buffer size */
    int     cnt;          /* valid samples in buffer */
    Float   *buf;         /* buffer */
    Float   *bufe;        /* buffer end */
    Float   *bufr;        /* buffer read pointer */
    Float   *bufw;        /* buffer write pointer */
    void    copy(Float *f, Float *t, int cnt);
};
```

File: circularbuffer.c

```
/*
 * Copyright (c) 1999-2000 AT&T Corp.
 * All Rights Reserved.
 */
#include "libcoder.h"
#include "circularbuffer.h"

CircularBuffer::CircularBuffer(int sz)
{
    buflen = sz;
    buf = new Float[buflen];
    bufe = &buf[buflen];
    flush();
}

CircularBuffer::~CircularBuffer()
{
    delete [] buf;
}

/* flush all data from the buffer */
void CircularBuffer::flush()
{
    bufr = bufw = buf;
    cnt = 0;
}

/* fill the buffer with all zeros */
void CircularBuffer::clear()
{
    int     i;

    bufr = bufw = buf;
    cnt = buflen;
    for (i = 0; i < buflen; i++)
            buf[i] = 0.;
```

```
}

/*
 * Save data in the buffer. Its legal to write more data to the buffer
 * than it can hold. In this case just the latest data is kept and the
 * read pointer is updated.
 */
void CircularBuffer::write(Float *f, int sz)
{
   int left;

   cnt += sz;
   do {
         left = bufe - bufw;
         if (left > sz)
               left = sz;
         copy(f, bufw, left);
         bufw += left;
         if (bufw == bufe)
               bufw = buf;
         sz -= left;
         f += left;
   } while (sz);
   /*
    * if more data has been written than can fit,
    * update the read pointer so it reads the latest data.
    */
   if (cnt > buflen) {
         cnt = buflen;
         bufr = bufw;
   }
}

/* retrieve data from the buffer */
void CircularBuffer::read(Float *f, int sz)
{
   if (sz > cnt)
         ::error("CircularBuffer::read: read too large");
   cnt -= sz;
   int c = bufe - bufr;
   if (sz < c) {
         copy(bufr, f, sz);
         bufr += sz;
   } else {
         int c2 = sz - c;
         copy(bufr, f, c);
         copy(buf, &f[c], c2);
         bufr = &buf[c2];
   }
}

/*
 * return the first sz samples at the head of
 * the buffer without modifying the buffer
 */
void CircularBuffer::peekhead(Float *f, int sz)
{
   if (sz > cnt)
         ::error("CircularBuffer::peekhead: not enough data");
   int c = bufe - bufr;
   if (sz <= c)
         copy(bufr, f, sz);
   else {
```

```
		copy(bufr, f, c);
		copy(buf, &f[c], sz - c);
	}
}

/* replace the first sz samples at the head of the buffer */
void CircularBuffer::replacehead(Float *f, int sz)
{
	if (sz > cnt)
		::error("CircularBuffer::replacehead: not enough data");
	int c = bufe - bufr;
	if (sz <= c)
		copy(f, bufr, sz);
	else {
		copy(f, bufr, c);
		copy(&f[c], buf, sz - c);
	}
}

/*
 * return the last sz samples in the tail of
 * the buffer without modifying the buffer
 */
void CircularBuffer::peektail(Float *f, int sz)
{
	if (sz > cnt)
		::error("CircularBuffer::peektail: not enough data");
	int fromstart = bufw - buf;
	if (sz > fromstart) {
		int c = sz - fromstart;
		copy(bufe - c, f, c);
		f += c;
		sz -= c;
	}
	copy(bufw - sz, f, sz);
}

/* replace the last sz samples in the tail of the buffer */
void CircularBuffer::replacetail(Float *f, int sz)
{
	if (sz > cnt)
		::error("CircularBuffer::replacetail: not enough data");
	int fromstart = bufw - buf;
	if (sz > fromstart) {
		int c = sz - fromstart;
		copy(f, bufe - c, c);
		f += c;
		sz -= c;
	}
	copy(f, bufw - sz, sz);
}

void CircularBuffer::copy(Float *f, Float *t, int cnt)
{
	for (int i = 0; i < cnt; i++)
		t[i] = f[i];
}
```